Programming
Research
Laboratory

BiGUL's Bidirectionality

Zhenjiang Hu & Josh Ko
National Institute of Informatics, Japan

Oxford Summer School on Bidirectional Transformations
27 July 2016

Plan

- Review well-behavedness, with partiality explicitly represented.

- Get a taste of putback-based design by looking at several BiGUL constructs.

- Start with a natural put semantics, and then revise it so that a corresponding get exists.

(Well-behaved) Lens

```
put :: s -> v -> Maybe s
get :: s      -> Maybe v
```

data Maybe a = Nothing | Just a

PutGet

get (put s v) = v

put s v ≈ Just s' ⇒ get s' = Just v

GetPut

get s = Just v ⇒ put s v = Just s

get (put s v) ⊑ v

Uniqueness of get

For any lenses l and r,

```
put l = put r  ==>  get l = get r
```

Proof

STS: get l s = Just v ⟺ get r s = Just v

get l s = Just v

wb l

⟺ put l s v = Just s

⟺ put r s v = Just s

⟺ get r s = Just v □

Replace :: BiGUL v v

```
put Replace s v = Just v
```

PutGet : get Replace v = Just v

GetPut :

Skip :: BiGUL s ~~v~~ (?)

()

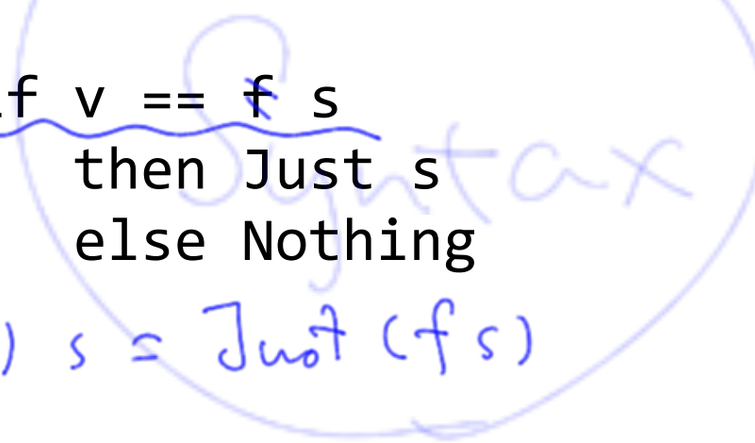
```
put Skip s v = Just s
```

```
get Skip s = Just ()
```

```
Skip :: (s -> v)
     -> BiGUL s v
```

```
put (Skip f) s v = if v == f s
                   then Just s
                   else Nothing
```

get (Skip f) s = Just (f s)



```
Prod :: BiGUL sl vl
     -> BiGUL sr vr
     -> BiGUL (sl,sr) (vl,vr)

put (l `Prod` r) (sl, sr) (vl, vr) = do
  sl' <- put l sl vl
  sr' <- put r sr vr
  return (sl', sr')

get (l `Prod` r) (sl, sr) = do
  vl <- get l sl
  vr <- get r sr
  return (vl, vr)
```

PutGet for Prod

Assume

```
put (l `Prod` r) (sa, sb) (va, vb)
   = Just (sa', sb')
(do sl' <- put l sa va
    sr' <- put r sb vb
    return (sl', sr')) = Just (sa', sb')
```

Prove

```
get (l `Prod` r) (sa', sb') = (va, vb)
(do vl <- get l sa'
    vr <- get r sb'
    return (vl, vr)) = Just (va, vb)
```

PutGet for Prod

```
(do sl' <- put l sa va
    sr' <- put r sb vb
    return (sl', sr')) = Just (sa', sb')
```

```
∃ sl'. put l sa va = Just sl' ∧
∃ sr'. put r sb vb = Just sr' ∧
Just (sl', sr') = Just (sa', sb')
```

```
put l sa va = Just sa' ∧ put r sb vb = Just sb'
```

PutGet for Prod

Assume

```
put l sa va = Just sa’
put r sb vb = Just sb’
```

Prove

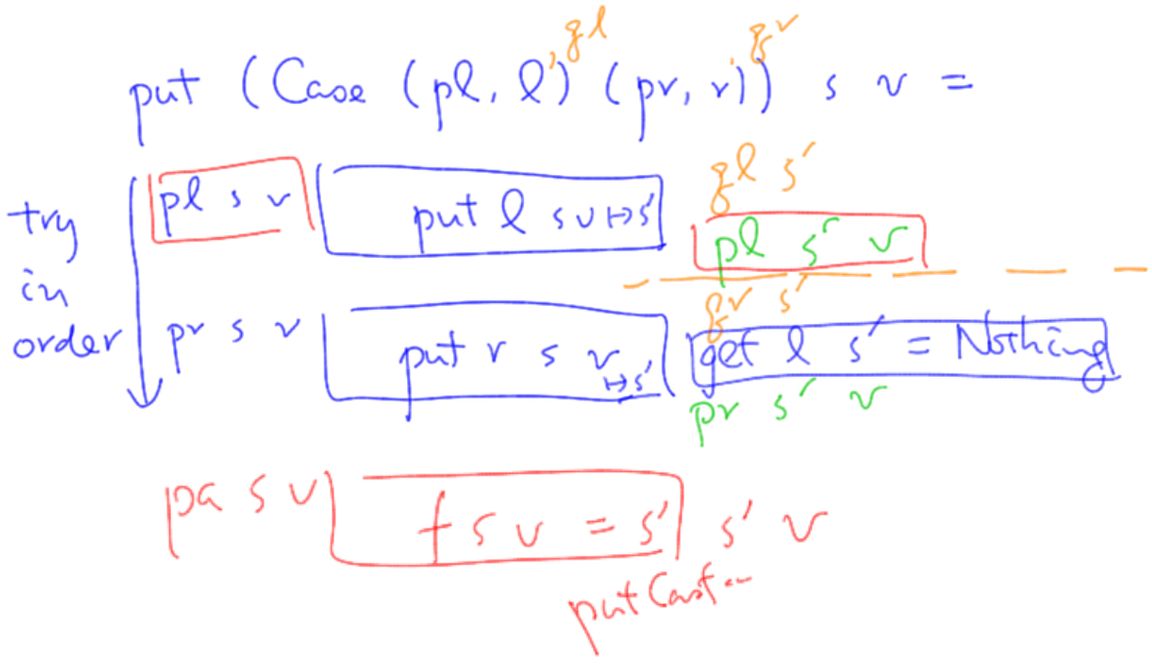
```
get l sa’ = Just va
get r sb’ = Just vb
```

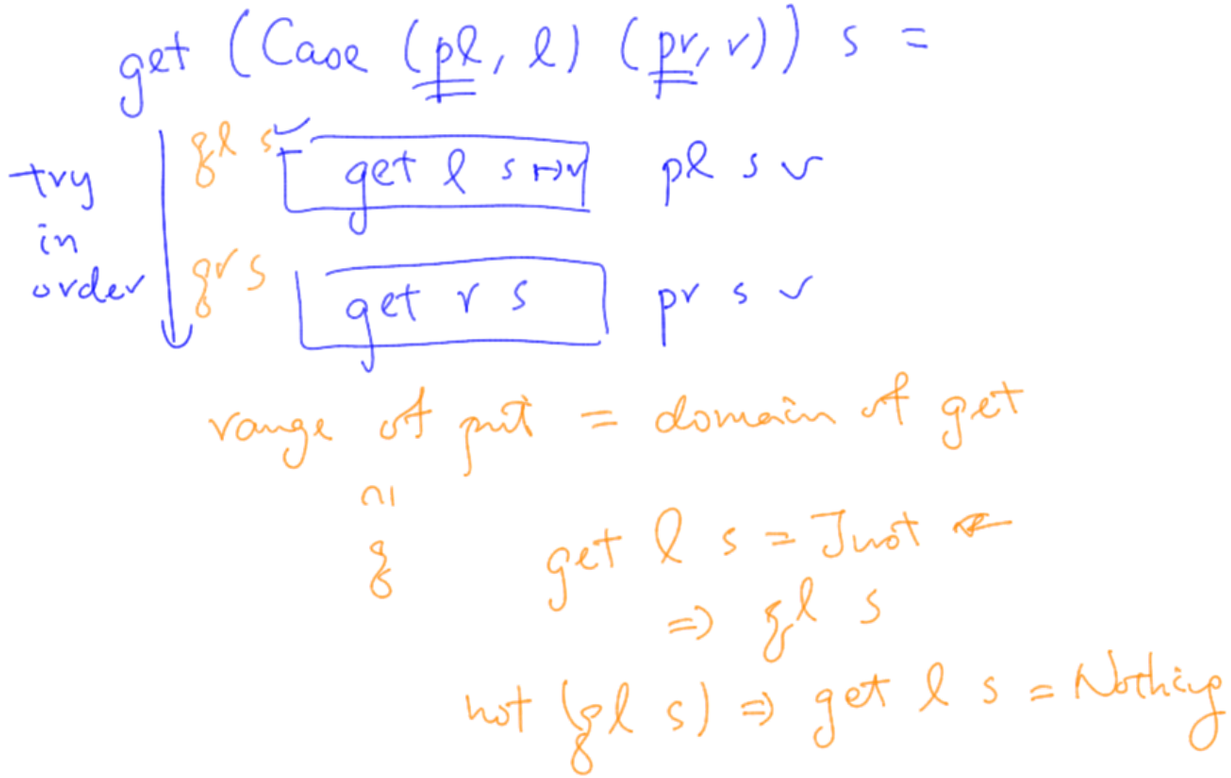
Case (binary)

```
type CaseBranch s v = (s → v → Bool, BiGUL s v)

Case :: CaseBranch s v
     -> CaseBranch s v
     -> BiGUL s v

put (Case (pl, l) (pr, r)) s v =
  if      pl s v then put l s v
  else if pr s v then put r s v
  else Nothing
```





Case (with exit conditions and adaptation)

```
type CaseBranch s v =
  (s → v → Bool, BiGUL s v, s -> Bool)

type CaseAdaptiveBranch s v =
  (s -> v -> Bool, s -> v -> s)

Case :: CaseBranch s v
     -> CaseBranch s v
     -> CaseAdaptiveBranch s v
     -> BiGUL s v
```

Rearrangement

- `$(rearrV [| \(Left (x,y)) -> (y,x) |]) …`
- Key: pattern matching and expression evaluation are invertible
- Type-safe implementation with Haskell’s generalised algebraic datatypes